Attacks and Risk Analysis for Hardware Supported Software Copy Protection Systems

Weidong Shi[†] Hsien-Hsin S. Lee[†‡] Chenghuai Lu[†] Tao Zhang[†]

College of Computing[†]
School of Electrical and Computer Engineering[‡]
Georgia Institute of Technology
Atlanta, GA 30332-0280
{*shiw, leehs, lulu, zhangtao*}*@cc.gatech.edu*

ABSTRACT

Recently, there is a growing interest in the research community to use tamper-resistant processors for software copy protection. Many of these tamper-resistant systems rely on a specially tailored secure processor to prevent, 1) illegal software duplication, 2) unauthorized software modification, and 3)unauthorized software reverse engineering. The published techniques primarily focused on feasibility demonstration and design details rather than analyzing security risks and potential attacks from an adversary's perspective. The uniqueness of software copy protection may lead to some potential attacks on such a secure environment that have been largely ignored or insufficiently addressed in the literature. One should not take security for granted just because it is implemented on a tamper-resistant secure processor. Detailed analysis on some proposed ideas reveal potential vulnerability and attacks. Some of the attacks are known to the security community, nevertheless, their implications to software copy protection are not well understood and discussed. This paper presents these cases for designers to improve their systems and circumvent the potential security pitfalls and for users of such systems to be aware of the potential risks.

Categories and Subject Descriptors: C.0.X [General]: Hardware/software interfaces

General Terms: Security

Keywords:tamper resistance,copy protection,attack

1. INTRODUCTION

Recently, there is a growing interest of designing secure processor architectures to provide a secure software execution environment on unprotected computing platforms. Such secure processor architectures adopt new features that support tamper resistance inherently. Coupled with cryptographic schemes, the secure processor architectures can be used to enable a secure environment where only authorized and un-tampered applications can be executed. Security is achieved by protecting data located inside the untrusted external hardware devices, mainly the off-chip memory and hard disks, by means of encryption and dynamic integrity checking [10, 16, 17, 18, 19, 11]. The fine-grained on-demand integrity checking and decryption mean that only when the data and instructions are brought into the tamper-resistant processor will they be decrypted and their integrity be verified. When executing applications, the architecture ensures that sensitive data and instructions of the applications will not be disclosed at any time and the software integrity is always guaranteed. The secure processor architectures can be used not only to prevent possible software-based attacks on protected applications, but also to prevent many hardware attacks which have not been addressed by any other similar secure systems [7]. Due to the strong protections provided, the secure processor architectures are able to address many security issues that have been haunting computer industry for decades, e.g. software copy protection, virus protection and trusted computing.

Although software copy protection based on tamper resistant processor is a promising direction for enforcing software right protection, there are still several remaining issues to be resolved before it can be accepted by the industry. These issues include, software testing issues, compatibility issues, programming model issues, privacy issues, performance issues, security issues, and etc.

Historically, computer crackers/hackers tend to be well-knowledged, highly motivated and, sometimes, well supported financially to attack computing platforms. Systems with weak security protection can be broken by sophisticated hackers as evidenced by the case of XBOX security key leak [8, 3]. Therefore, it is crucial to perform detailed investigations on the security of tamper-resistant processor architectures so as to ensure that the proposed designs can meet high security standards. The security analysis on secure processor architecture is twofold — the generic cryptanalysis; and the security analysis from the architectural perspective. The techniques for generic cryptanalysis are well-known to the security community. Consequently, such analysis is relatively easy to perform. However, processor architectures and software copy protection also inherently

DRM'04, October 25, 2004, Washington, DC, USA.

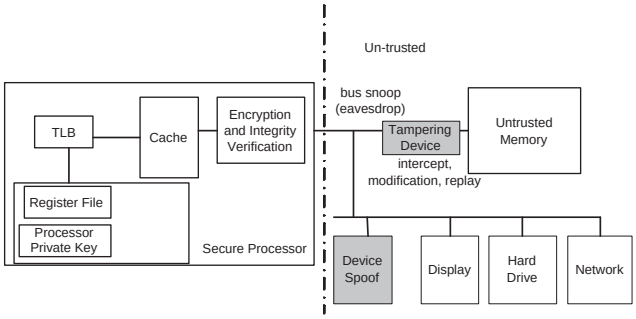

Figure 1: Secure computing model

possess some unique characteristics which could have security impacts. Examples of such characteristics are program traces, program branches, the patterns among instruction sequences and etc. Those characteristics must receive serious attention while evaluating the security on processor architectures. This paper particularly focuses on the security issues from the architectural perspective for proposed designs and analyzes the security pitfalls of secure processor architectures from an adversary's perspective. The goal is to provide useful and valuable insights to the system designers. Note that awareness of the potential attacks can lead to much more secure solutions and avoid security pitfalls. Besides, users of the secure processor will also benefit by knowing the potential weakness and risks.

The rest of the paper is organized as follows. In the next section, we briefly address some general security issues associated with tamper resistant/copy protection system and the basic assumptions made by most proposed solutions. It also presents some copy protection models that are used as example systems for the succeeding attack analysis. Then, in section 3, detailed explanation of various potential risks are presented where each attack is discussed. Section 3 is followed by analysis of implications of the presented attacks and suggestions for strengthening security. The last section concludes the paper.

2. HARDWARE ATTACKS ON SOFTWARE COPY PROTECTION

A typical secure processor architecture model consists of a tamper-resistant/copy protection processor, external memory and peripherals as shown in figure 1. Naturally, the protection boundary is drawn between the processor and the external hardware units. Hardware units, like registers and on-chip caches, are protected from any possible attack while the remaining of the hardware units such as the external memory and peripherals are considered vulnerable to physical attacks. Besides the aforementioned hardware, the secure computing model also includes a small trusted program, e.g., the XVMM in XOM and the secure kernel in *Aegis* [10, 16]. The trusted program will be called the secure kernel hereafter. The secure kernel runs at a higher privilege level than any other program including the regular operating systems and is responsible for performing encryptions/decryptions for the protected applications when their data are crossing the protection boundary. The secure kernel is also responsible for protecting sensitive data that resides in the private memory and registers during context switches.

Some common features of software copy protection solutions based on tamper resistant systems are described as follows.

- It is assumed that everything outside the processor is unprotected and subject to potential malicious tampering. The physical RAM itself is unprotected and hackers could read/overwrite the memory content directly without involving the processors. Furthermore, all the system/peripheral bus traffic is exposed and could be traced by the hackers.
- Like other tamper resistant systems, there is a pair of public-private keys associated with each secure processor. The secure processor's private key is permanently burnt inside the processor core and can not be accessed by any software [10].
- There is hardware supported encryption/decryption and integrity check. When an instruction or data cache line is brought into the secure processor, it is decrypted and integrity of the entire virtual memory space is verified using a hash tree or MAC tree [15, 1]. When a cache line is evicted from the secure processor, it is encrypted and the hash/MAC tree is updated. The keys used for encryption/decryption and integrity verification are given by the software vendors and encrypted by the secure processor's public key.

Most of the proposed systems support separate protection on software confidentiality and integrity. Some systems also support selective protection where either sections of application software or slices of application program are protected [19].

Wherever there is copy-protection measure, there will be attacks. Computer hackers often have many techniques at their disposal to crack out the secret designers try to hide in either software or hardware. If necessary, they can construct specialized hardware or even specially designed printed circuit boards or cracking/spoofing machine to recover the protected secret. Their efforts and dedication should never be underestimated. When comes to the issue of copy-protection and software confidentiality, the problem becomes even harder. The entire protection on software confidentiality is broken if an adversary compromises a single copy of the protected software. In this paper, we study the security issues on processor architecture from an adversary's perspective. In consequence, we present possible attack scenarios for cracking a piece of secure software executed on a hardware-facilitated tamper resistant/copy protection environment. Also note that the attacks are applicable to other forms of secure processor architectures as well, such as TCPA [7].

There are some typical hardware techniques for breaking software copy protection, including software execution trace analysis, mod-chip spoofing devices, and machine emulator.

- **Trace analysis.** Adversary can collect information of protected software through logging and analyzing software execution traces. Many programming primitives even in encrypted form are highly predictable from observation of program execution traces. For examples, conditional/unconditional branches can be easily identified and traced even after they are encrypted. Index and counting variables that are widely used in programming may also be recovered with the assistance of program traces. For instance, if a program initializes a large array, loop and other related variables can be recovered by

analyzing the memory access pattern. As to the feasibility of tracing front side bus traffic, it could be achieved by using an interposer board [9] and multi-channel high frequency logic analyzer. As shown by [8, 3], skillful computer hackers may build custom tracing devices using cheap and available commodity components. Program traces also provide feedback to the adversary after changes are made to the protected software.

- **Mod-chip.** Here we use the word mod-chip to refer to all the spoofing devices designed for unraveling copy protection mechanism. Mod-chip can be used to record and replay memory and bus transactions. They can also be designed for hijacking another device's signal or launching device spoof attacks.
- **Emulator.** Software copy protection is deemed broken if the protected software can be executed on a machine emulator without authorization. It is very difficult to fight against this attack without using software encryption. Protection on software confidentiality can prevent a machine emulator from breaking protection on software rights.

3. RISKS ANALYSIS OF CERTAIN DESIGN CHOICES

A number of software copy protection solutions based on tamper resistant processors have been proposed recently. Most of them share similar assumptions and features presented in the previous section. In XOM [10], a per-process encryption key (triple-DES) is used to decrypt software on the fly, while Aegis [16] system uses AES. One major difference between Aegis and XOM is that Aegis employs an on-chip hash tree to verify the integrity of the entire process space in execution time, thus preventing a memory replay attack. As studies have indicated, block-cipher-based systems can incur non-trivial performance penalty for benchmark suites [16, 18]. Systems using counter-mode encryption and relaxed integrity checks [17, 18] are proposed because they accelerate software execution. Alternative solutions for better performance have been proposed, such as encrypting only small amounts of carefully selected instructions, dubbed software slices [19]. 1 lists some of the systems and their differences.

Although many systems have been proposed, very few studies were performed on security analysis and risk assessment of tamper resistant processor based copy protection systems. Most of the documented systems rely on certain assumptions where security is taken for granted. Here we will examine some of these assumptions and their security implications. The purpose is to gain meaningful insight on the security of tamper resistant/copy protection systems. It is important to point out the difficulty of analyzing proposed tamper-resistant processors. Since they are not real systems, many times, interpretation of a design could be subjective. The goal of this paper is security analysis based on documented design ideas instead of comparing or evaluating the robustness of these proposed systems.

3.1 Lazy Authentication and Counter-mode Encryption

By "lazy" integrity checking, we mean that either integrity of instructions are not verified promptly on a in per-instruction basis or machine state (memory/processor) can be altered before per-instruction integrity checking is completed. "Lazy" integrity check also refers to the situation where the data source is used as an operand and the result of computing using un-authenticated data is allowed to modify the processor state before its integrity is authenticated. Techniques for "Lazy" integrity check are proposed for their performance advantage over more rigorous, secure, and timely integrity checking.

Opcode	Number			
Opcode	RA	Disp		
Opcode	RA	RB	Disp	
Opcode	RA	RB	Function	RC

Figure 2: Alpha Instruction Format

Many modern processors such as Alpha, MIPS, and ARM embraced RISC design philosophy. The simplicity of RISC instruction set allows processor designers to perform more aggressive instruction fetching/decoding and pipelining. However, the regular format and simplicity of RISC instruction set also eases an adversary's effort to guess encrypted RISC instructions. In subsequent sections, we will use Alpha instruction set as an example to illustrate how an adversary can exploit the simplicity of Alpha instructions to crack instructions encrypted using counter-mode. The attack requires that the adversary is able to obtain front side bus traces of program execution.

The vulnerabilities of the RISC instruction set are, 1) all the instructions have the same length and in many cases they are short, 16 bits, 24 bits, or 32 bits. One weakness of short instructions is that it may be vulnerable to brute force attacks; 2) the instructions are well formatted (e.g. fixed opcode field) for reducing the complexity of decoding logic. For example, bit[31:26] of Alpha ISA is fixed as the opcode. As shown in the example below, one risk posed by this RISC property is that it may allow incremental guessing of instructions. In such an attack, the adversary divides each instruction into portions (opcode, operand one, operand two, and etc.) and launch brute-force guess piece by piece on each portion of the targeted instruction. This significantly reduces the search space of a brute-force attack. For instance, to apply brute force attack on a 32 bit instruction, there are 2^{32} possibilities. However if the instruction could be divided into four 8-bit portions and each portion could be exhaustively attacked, only 4*256 brute-force trials are needed, way smaller than 2^{32}; 3) RISC philosophy advocates a small set of instructions instead of a large number of complex instructions. This also reduces the search space of brute-force attack. Figure 2 shows some of the Alpha instruction formats used in the example attack [5].

To make it easier to understand, assume that the targeted Alpha binary does not use any dynamically linked libraries, all the instructions are packed into one code section, and each instruction is encrypted using counter-mode (regardless how the pseudo-random one-time-pad is generated). Assume also that the adversary has no priori knowledge of the program but is able to obtain front-side bus traces of program execution. Further assume that the secure processor only performs a "lazy" integrity checking on executed instructions.

Table 1: Some Tamper Resistant/Copy Protection Systems

System	Run Time Integrity	Confidentiality	Range
XOM [10]	hash	triple-DES	instructions and data
Aegis [16]	hash tree	AES	instructions and data
Yang, Zhang, and Gao [18]	not specified	DES counter mode	instructions and data
LogHash [17]	log hash	AES counter mode	instructions and data
Zhang,Gupta [19]	not specified	not specified	instruction slices

In order to launch the attack, the adversary has to start from something s/he already "knows", even though the codes are encrypted. One candidate would be invariant instruction sequence that is pretty much fixed in almost all the executable images generated by a compiler. For example, almost all the benchmarks in the compiled SPEC2000 binary have the same startup prologue instructions as the follows,

```
lda sp,-16(sp)
stq zero,8(sp)
...
```

Starting from these two known instructions, the adversary could launch known-plaintext attack and crack out more instructions that s/he could not guess so easily. A few good candidates would be the instructions in the middle of the code section. Assisted with the two known instructions, the adversary could control the next executed instruction by modifying the known instruction into a jump instruction with any target address. We use a sample code from benchmark 186.crafty as an example in table 2.

The code section contains 37,789 instructions. The list shows one candidate instruction (addq) in the middle that the adversary may choose as the jump target. The reason to choose instructions in the middle is because as shown later, they, after being altered into jump instructions by changing the opcode, are more likely to jump into valid code space than instructions close to the boundaries. Next, the adversary may perform a brute-force attack on the opcode of the targeted instruction by changing it. Since the Alpha instruction always uses bit[31:26] as opcode, the adversary could figure out bit[31:26] of the one-time-pad with at most 64 trials of opcode guessing. The opcode of the targeted instruction "addq" is 0x10. Assume that the adversary's first guess is opcode 0x4. The speculated bit[31:26] of the one-time-pad would be 0x21. Then s/he could change the instruction into an unconditional jump by altering bit[31:26] of the targeted instruction to the result of 0x21 $\oplus$ 0x30, where 0x30 is opcode of jump. Because the opcode guess is wrong, the altered instruction will be decrypted into an AND (0x11) instruction instead of a jump. Since there is no jump in the trace of instruction fetch, the adversary is certain that the guessed opcode is incorrect. Assume that the next opcode guess is 0x10 and is correct. This time, the program trace will show jump of program execution to target address 0x1200263E0 from address 0x12001139c(0x12001139c + 0x5411*4). This will reveal bit[20:0] of the encrypted target instruction. The rest 5 bits (bit[25:21]) could be guessed by trying to alter the targeted instruction into a FETCH instruction, where all the 5 bits should be zero for a valid FETCH instruction. At most another 32 trials are required. Note that the above opcode attack can be launched in parallel using multiple machines with each machine taking one alternative guess. Given a moderate size of 64 machines, only under two parallel trials, the adversary is able to crack the encrypted target instruction.

The above case represents an ideal situation where the altered instruction jumps to an address within the code section. Since the displacement field of jump instruction has 21 bits, it is very likely that the targeted address may be outside the range of the current code segment. The adversary could tackle this problem in two possible ways, 1) modify the virtual address to physical address translation table so that the targeted address would be translated and fetched. A rigorous integrity checking and protection on TLB (translation lookup buffer) and process context of address translation may prevent such an attack; 2) Brute force attack on both the opcode and the remaining displacement field bits whose range is outside the code segment. In the above benchmark example, since there are about 37,000 instructions, only the high 6 bits of the total 21 bits of the displacement field need brute force guessing. Given 64 machines with each machine taking one guess of the opcode, at most 64 parallel trials are sufficient to break both the opcode and the remaining high bit of the displacement field given the assumption that there is no alternative way for the adversary to tamper the address translation mechanism.

Many embedded systems do not use virtual physical address translation. For those systems, the tampered addresses could be observed directly on the bus. Since fetching the next instruction could be started before execution of the previous instruction is completed, even stronger instruction authentication is required. In such systems, jump targets of conditional and unconditional branches should not be fetched before integrity of the jump instructions is verified.

An adversary may use the above technique to figure out all the instructions. Alternatively, if only program code is encrypted, the adversary can use the following "short-cut" procedure. Firstly, s/he can figure out a short sequence of instructions (about 40) in the middle of the code section using method described above. Then s/he can speed up the attack by trying to alter other encrypted instructions into a STORE instruction. Take the mov instruction in 2 as one example. Through brute-force attack on the opcode, the mov instruction could be altered into, stw a12,a12(1043).

To crack out the remaining 26 bits, the adversary may firstly transfer execution to the short code sequence which s/he has figured out, load a constant value to all the 32 Alpha registers by altering the cracked 40 instructions so that the computed data address (address register value + displacement) would be certainly within the space of data virtual address translation, then s/he can transfer execution to the altered targeted instruction. All these can be completed using less than 40 altered instructions. By observing the traces of data access, s/he would be able to figure out the last 16 bits of the targeted instruction (0x0413). This requires only one parallel trial or 64 single trials. To figure

Table 2: List of Crafty Code Section

Address	Plaintext	Ciphertext	Instruction
0x120008840	0x23defff0	0x3127d04a	lda sp,-16(sp)
0x120008844	0xb7fe0008	0x4c0d4ef4	stq zero,8(sp)
...			
0x120010194	0x46520413	0xa0481bf0	mov a2,a3
...			
0x12001139c	0x40c05411	0x9426814a	addq t5,0x2,a1
...			
0x12002d670	0x23de0010	0x3704e241	lda sp,16(sp)
0x12002d674	0x6bfa8001	0x7a3250bf	ret

out bit[25:16], the adversary may repeat the same procedure. But instead of loading the same constant to all the Alpha registers, s/he will load a unique value to each Alpha register. Since the displacement is already known, subtracting the displacement from the write address observed from the memory trace will reveal one unique value loaded to the alpha registers. This unique value will tell which alpha register is used and its register ID reveals 5 bit plaintext of bit[25:16]. The unique value stored will tell which register is used as data source and its register ID reveals the remaining 5 bits of bit[25:16]. In total, only two parallel trials are sufficient to crack the mov instruction.

As shown above, with only an amortized cost of two parallel trials/per instruction using 64 machines, the adversary is able to figure out a counter-mode protected program within a reasonable time. Assume that it takes 30 seconds for the adversary to complete one parallel trial (in fact, this is an overestimate and the real time needed could be much less after the procedure is automated). It takes only about one and half month to crack out a program with about 64K instructions (256K code size).

We call the described attack technique, *"alter then trace attack"* (ATT attack). To use this attack, 1) the adversary must be able to alter a piece of software (program or data) bit-by-bit (satisfied by all the "counter mode" based protection); 2) altered instructions can be executed and integrity of executed instructions or used data is not verified promptly or rigorously on per-instruction basis.

In this section, we show only one way of exploiting the simplicity of RISC instruction set. Other methods based on the same principle could be "invented" by a creative adversary. In short, all the "counter-mode" based approaches with "lazy" authentication check are potentially vulnerable. Approaches that check integrity in a timely fashion but have other flaws may also be vulnerable when certain conditions are met such as when the adversary is able to tamper the address translation or brute force attack on the integrity code. Note that the risk of "lazy" authentication can be reduced by using hardware based *control flow/memory fetch address obfuscation*, such as the techniques proposed in [21, 20]. However, the hardware complexity and overhead can be high. It should be pointed out that software based information hiding technique such as [14] could not prevent the discussed ATT attack.

3.2 Software Slice

Since the cost of whole program protection is really high and unacceptable for certain applications, solutions that protect only portions of an application are also proposed [19]. The rationale is that an application can be equally protected if the small protected portions are impossible for an adversary to recover/copy and at the same time they are absolutely necessary for the correct execution of the protected application. A concept, called software slice, is introduced to refer to a trace of dependent program instructions. Sometimes, traces of program can contain complicated operations such as boolean, relational, high degree arithmetic operations, or conditionals (branches or program control flow). A program trace with more complex operations and control flows is assumed to be more difficult to recover than program trace of simple arithmetic operations. It is assumed that protection of only small number of program traces or slices with high complexity is sufficient to protect the whole software. The complexity of a program slice is often measured in terms of length of the trace, number of high degree arithmetics, number of boolean and logic operations, number of conditionals, and etc. For example, complexity can be defined according to the partial order constant $<$ linear $<$ polynomial $<$ rational $<$ arbitrary (boolean and logic operations). Furthermore, control flow is considered to have high complexity. It assumes that traces with boolean operations or conditionals are harder to crack than traces with only simple arithmetic operations. Protected software slices (protected component) are encrypted, stored and executed on a security co-processor while the rest of the program (open component) is executed on a regular un-protected processor. Communication between them is not protected.

Software slice based protection is a very interesting technique because of its potential connections with some other research areas in computing. One interesting question about software slice is whether its security can be more rigorously defined using a metric other than a simple count of complex operations and conditionals. From an adversary's perspective, s/he does not have to recover the exact original code sequence to break the security. As long as s/he can come up a program that is functionally equivalent to the protected program trace, it is enough. Since nothing else is protected except the program slices, the adversary can treat the protected codes as a black box and try to come up with another code sequence that can emit the same output under the same input. This is obviously a computationally hard problem if the goal is to design a general approach for emulating any random program black box. However, the adversary may not be interested in designing such a universal approach that can model any program traces/slices. S/he may be only interested in breaking some or maybe only one trace/slice that matters to her/him in reasonable time. To answer the questions of security strength of the software

```
f(float x, float y, float z, bool b)
{
    if (x<3.5 && b==true)
    z = x*y;
    else
    z = 2*x*x;
}
...
```

Figure 3: Example Program

slice approach, some related research areas may be able to give some insight,

Automatic programming and state machine inference using examples. The goal of automatic programming or software synthesis using examples is to ask the computer to generate desired software based on specified input/output data. Some of the early work includes inferring lisp program using examples [2]. The theory of inductive inference has been one of the foundations for automatic programming. It appears that the task of compromising a software slice is similar to the task of synthesizing example-based programs. However, an adversary has more restrictions recovering a protected software slice than s/he does in the situation of software synthesis, in which more information can be provided to guide the automatic programming. State machine inference and automata learning based on input/output is also closely related to software slice security.

Neural-networking modelling. Neural-networking has the power for universal data interpolation/approximation. It is known that neural network using rational modelling neurons is able to interpolate or approximate multi-dimensional contiguous functions. In [6], a unique technique that combines data clustering and neural networking is presented to approximate functions that are piecewise contiguous. For example, the code sequence in figure 3 may be modelled using techniques derived from [6] .

Furthermore, binary neural network [4] is known to be very powerful to model binary function and finite automata. The question is the time/space complexity of using binary neural network to model a specific program slice. Properties of binary neural network has been well studied. A theoretical concept called, *linear separability* [13], is introduced to measure the complexity of a binary function that is to be modelled by a binary neural network. According to [12], time and space requirement to model a function is proportional to *linear separability*. In reality, it would be hard to give an estimate of *linear separability* for any given program slice. However, this concept may give some insight about the complexity of program traces/slices against potential machine learning based attack.

The topic of the theoretical strength of program traces/slices is an interesting problem and could become a shared subject by the area of security, theory, and machine learning.

3.3 Active Information Flow Attack

Some proposed systems use the same encryption scheme and the same key for protecting both software instructions and dynamic data. Furthermore, to reduce potential performance overhead, many systems expose protection control to the users and allow users to configure which section of software instructions or data should be protected. For instance, new instructions can be proposed to allow fine granularity control of confidentiality protection. 3 lists four such instructions and their corresponding semantic.

Instructions or code images desired to be kept as secret are encrypted and enclosed by a pair of enter_security and exit_security statements. Instructions outside the protected domain are executed as usual. Similarly, data can be exchanged between a tamper-resistant processor and the untrusted memory using either regular load/store or protected secure load/store. The difference is that only under secure load/store will data be decrypted/encrypted and authenticated. Introducing these fine-grained security instructions on one side, improves the flexibility and programmability of security control but on the other side, it increases the difficulty of security protection. One consequence of fine control on security protection is that some pieces of information are in the protected domain while some others are not. The boundary between them is often not set by the pair of enter_security and exit_security statements but is actually determined by the flow of information. Using secure_load and secure_store as examples, although both instructions are executed inside a protected code region enclosed by enter/exit_security, the actual data source and store/load address may come from either protected memory or unprotected memory. This property may attract attention of an adversary. Although it is hard to give a complete picture about the implication of exploits based on information flow or even enumerate all the possible breaches, there is a likelihood that an adversary could break a protected application under certain conditions.

Consider the simple code in figure 4. The piece of code is a simple example that shows that the actual security boundary is not set by the pair of enter_security and exit_security. In fact, every secure_store or secure_load could become a boundary between protected and unprotected information. In real world scenarios, the flow of information can be far more complicated than this simple example. To decide whether a secure_load/store uses address or data from un-protected world is not a simple matter as it requires complicated information flow analysis such as memory disambiguation. For the given example, because array_dat is not protected, an adversary can replace it with a short attacking code sequence such as the one shown in figure 5. After the code sequence is encrypted, a pair of enter_security and exit_security can be inserted to the array_dat to enclose the encrypted attacking code. Now, the adversary has a piece of encrypted malicious code that is ready to use. If the code space can be overwritten, the adversary can patch/alter encrypted code by supplying a pointer to the encrypted code as output address to secure_store.

To give another example that is purely data based, consider that there is a link-list holding computed results that need to be disclosed from protected memory to unprotected memory, see figure 6. A real world scenario could be that a user code (unprotected) wants data stored in a link-list to be processed by some protected program (encrypted). To hide the internal secret, the protected program may compute the data in secret mode using secure_store every time data of each node is updated. After the computing is completed, the protected program discloses the results by traversing the link-list and converting each data element using secure_load followed by a regular_store that stores the data to a place the

Table 3: Example Security Instructions

Instruction	Format	Definition
`enter security`		Start security mode after this instruction. A key is used to decrypt all the succeeding instructions
`exit security`		Exit security mode. No decryption for succeeding instructions
`secure store`	sd $rt,offset($base)	Stores $rt into memory [$base+offset] (data encrypted with the same key used for decrypting codes)
`secure load`	ld $rt,offset($base)	Load $rt with memory [$base+offset] (data decrypted using the same key used for decrypting codes)

```
// not protected data and code
unsigned int array_dat[] = { ... };
...
//protected code,encrypted/authenticated
enter_security
...
// load array_dat and secure_save
unsigned int x;
for (i=0; i<sizeof(array_dat)/4; i++)
{
     load array_dat[i] to x;
     secure_store x to array_dat[i];
     ...
}
...
exit_security
```

Figure 4: Security Boundary Is Set by secure_load/store

```
...
secure_load some secret
save secret to some unprotected memory
...
```

Figure 5: Attacking Code Sequence Used to Replace array_dat

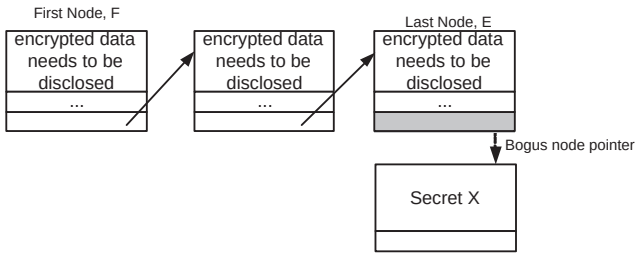

Figure 6: Compromising Secret X Using Flow Attack

```
// disclose results of computing to public
struct node_t {
     unsigned int dat;
     ...
     node_t* pnext;
}
//protected code,encrypted/authenticated
enter_security
... //process link-list
//release results
node_t* pnode = head of link_list;
while (pnode)
{
     secure_load pnode->dat to temp;
     save temp to un-encrypted memory;
     pnode = pnode->pnext; // regular load
}
exit_security
```

Figure 7: Disclosing Data in Link-list

user can access (figure 7). Assume that pointers that point to the next link-list node are kept in unprotected memory (accessed using regular load). Furthermore, there is a secret X hidden in the protected program that an adversary wants to recover. What the adversary can do is right before the protected program starts to disclose the link-list, s/he can modify the NULL pointer of the last node so that it will point to the secret X. This will cause the protected program to disclose secret X into unprotected space.

It is not an easy task to give a complete picture of risks associated with flow of information or even enumerate all the possible scenarios. However, this is surely an interesting problem. Techniques based on program analysis and compiler technique could be a direction for solutions. Alternatively, protection can be extended to the whole program and every load/store is protected.

3.4 Weak Cipher/Short Key

Sometimes, there are legitimate reasons to use short keys or simple ciphers, for example performance consideration or tight chip size constraints. However, weak ciphers and short keys should be avoided when possible. The nature of software execution provides many opportunities for an adversary to launch brute force attacks to break weak ciphers or short keys. To give a concrete example, considering the following code sequence protected with a short integrity code stored side by side with every 8 instructions (cache line size),

If the purpose of attack is to bypass the security check,

```
// Code Example Begin
push ax
push bx
push cx
push dx
push ex
call security_check
/*a jump to a subroutine */
tst ax, 0
/* assume return value in ax*/
bne security_failed
// Code Example End
```

Figure 8: Example Assembly Program

```
// Altered Code Example Begin
nop
nop
nop
nop
nop
nop
nop
nop
// Altered Code Example End
```

Figure 9: Altered Assembly Code

there are two brute-force modifications the adversary could launch, one on the integrity code itself and the other one on the code sequence. Firstly, to brute force attack on the short integrity code, the adversary could alter the code sequence to another sequence,

This new code sequence very likely will have a different integrity code from the unaltered version. To fool the authentication check, the adversary could try to execute the altered code each time with a different integrity code guess. If the integrity code is short, for example 16 bit long, after certain number of trials, a working integrity code could be found. The adversary is able to test whether a trial is successful through tracing of the program execution. If instruction fetch starts on the instruction after the last nop, the adversary knows that it is a success trial.

Alternatively, the adversary could come up a huge number of "equivalent" attacking code sequence and hope that one of them will have the same integrity code as the unmodified code sequence as shown in figure 10.

Different attacking code sequence will assign different random number to ax, bx, or cx. If the integrity code is short, by chance alone, some attacking code sequence will have the same integrity code as the unmodified version. This allows the adversary to replace the original code sequence with a new one without changing the integrity code.

Another reason that short key should be avoided is that they are vulnerable to so called "indexing/counting variable attack". Indexing/counting variables are indispensable parts of a program. In certain cases, such variables may count to very large number, for instance,

The risk associated with indexing/counting variable is

```
// Altered Code Example Begin
mov ax, random_num
xor ax, ax
mov bx, random_num
xor bx, bx
mov cx, random_num
xor cx, cx
nop
nop
// Altered Code Example End
```

Figure 10: Altered Assembly Code

```
// count a large number
while (1) {
 cycle_count++;
 ...
}
// Code Example End
```

Figure 11: Predictable Counters

that they are highly predicable and traceable even after they are encrypted. Recovery of certain indexing/counting variables may reveal immense amount of data to the adversary, sometimes maybe enough to perform an off-line crack of the encryption key if the key is short. Assisted with information from branch analysis and knowledge of how a particular compiler would generate codes for these variables, it would be not too difficult for the adversary to discover the memory locations of these variables. An alternative way for the adversary to discover counters that count to a large number is to find out most frequently updated memory locations ("hot spots") and investigate whether counting variables are held in these locations.

As to the feasibility of the attacks, one argument is that in today's processor, most instructions and data are cached inside the processor and their states are not visible to the outside world. However, if the adversary could tamper the memory management mechanism. S/he could mark certain memory range as un-cachable. Alternatively, most of today's processors support bus snoop based cache-coherence protocol. It is possible for an adversary to insert fake DMA (direct memory access) requests causing a processor to mark a cache line as shared and write the data back each time it is updated. Such attack involves building a device that is able to insert false coherent memory read to the system memory. One way of doing this is to use a cheap FPGA board designed for peripheral device development and load it with logic that keeps sending coherent DMA requests to the targeted memory address. Another way is physically disabling the cache. Most processors allow user to disable cache through resetting BIOS.

3.5 Design Recommendations

Based on the security analysis presented in this paper, it is recommended that a strong encryption algorithm and sufficient size key should be used. For stream ciphers and counter-mode based schemes, a combination of the following safeguards could be used to strengthen security, 1) to

avoid modification of codes, no instruction should be allowed to complete or modify processor/memory state before it is authenticated and its integrity is verified. For embedded systems that do not use virtual addressing, the instruction fetch using the decoded jump target address should not be issued before the jump instruction is authenticated. This will prevent many of the attacks mentioned above; "lazy" integrity check may enhance performance but at the same time could undermine the security protection and should be used with care when used together with counter mode encryption; 2) Both the TLB and process context of address translation should be well protected and authenticated. No translated address is allowed to be effective unless integrity of the translation is verified; 3) when fine-grained control of security protection is exposed to the users, additional techniques are preferred to prevent active information flow based attacks.

4. CONCLUSION

This paper presents a security analysis and risk assessment for software copy protection systems built on tamper resistant processors. Illustrated by examples, we show that the security of a secure processor supported copy protection system should not be taken for granted. More in-depth studies toward security issue for such systems are needed to develop a set of security guidelines for future designs of secure processor based copy protection systems.

5. ACKNOWLEDGE

We thank Jeff Lotspiech, who gave invaluable comments and helped polishing the presentation.

6. REFERENCES

[1] M-TREE: A Fast Secure Architecture for Protecting the Integrity and Privacy of Software. *Submitted for publication. http://www.cc.gatech.edu/people/home/lulu/Mtree.pdf*, 2004.

[2] A. W. Biermann. The inference of regular lisp programs from examples. In *IEEE Trans. on Systems, Man, and Cybernetics, 8(8):585-600*, 1978.

[3] Andrew "bunnie" Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, 2003.

[4] N.S. Chaudhari and A. Tiwari. Extension of Binary Neural Networks for Multi-class Output and Finite Automata. *Neural Information Processing: Research and Development. Ed. Jagath Rakjkapse and Lipo Wang*, December, 2003.

[5] Compaq Computer. *Alpha 21264 Microprocessor Hardware Reference Manual*.

[6] G. Ferrari-Trecate and M. Muselli. A New Learning Method for Piecewise Linear Regression . *International Conference on Artificial Neural Networks*, (ICANN02), 2002. Madrid.

[7] The Trusted Computing Group. https://www.trustedcomputinggroup.org/home. 2003.

[8] A. Huang. Keeping secrets in hardware the microsoft xbox case study. *MIT AI Memo*, 2002.

[9] Interposer. http://www.arium.com.

[10] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectual Support For Copy and Tamper Resistant Software . In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[11] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 178–192. ACM Press, October, 2003.

[12] M.Anthony. Boolean functions and artificial neural networks. *Boolean Functions: Volume II. Ed. Yves Crama and Peter Hammer*, December, 2003.

[13] M.Anthony and P.L.Bartlett. Neural Network Learning: Theoretical Foundations. In *Cambridge University Press*, 1999.

[14] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, 1999.

[15] E. Suh, B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Caches and Merkle Trees for Efficient Memory Authentication. In *Proceedings of the Ninth Annual Symposium on High Performance Computer Architecture*, February 2003.

[16] E. G. Suh, D. Clarke, M. van Dijk B. Gassend, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the Int'l Conference on Supercomputing*, 2003.

[17] E. G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings 0f the 36th Annual International Symposium on Microarchitecture*, December, 2003.

[18] Jun Yang, Youtao Zhang, and Lan Gao. Fast Secure Processor for Inhibiting Software Piracty and Tampering. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, December, 2003.

[19] Xiangyu Zhang and Rajiv Gupta. Hiding Program Slices for Software Security . In *Proceedings of the 2003 Internal Conference on Code Genration and Optimization*, pages 325–336, 2003.

[20] X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In *Proceedings of the International Conference on Compilers, Architecture, Synthesis for Embedded Systems*, 2004.

[21] X. Zhuang, T. Zhang, S. Pande, and H.-H. S. Lee. HIDE: Hardware-support for Leakage-Immune Dynamic Execution. Report GIT-CERCS-03-21, Geogia Institute of Technology, Atlanta, GA, November 2003.